Functional Reactive Programming

Part 4/4 : Microservices Architecture – Rx Java 2

Araf Karsh Hamid, Co-Founder / CTO, MetaMagic Global Inc., NJ, USA

ReactiveX is a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming

Part 1 : Microservices Architecture

Part 2 : Event Storming and Saga

Part 3 : Service Mesh and Kafka

Source Code GitHub: https://github.com/meta-magic/rxjava2

Architecture Styles

- HEXAGONAL ARCHITECTURE
- ONION ARCHITECTURE

Source : https://skillsmatter.com/skillscasts/5744-decoupling-from-asp-net-hexagonal-architectures-in-net

Hexagonal Architecture

Ports & Adapters

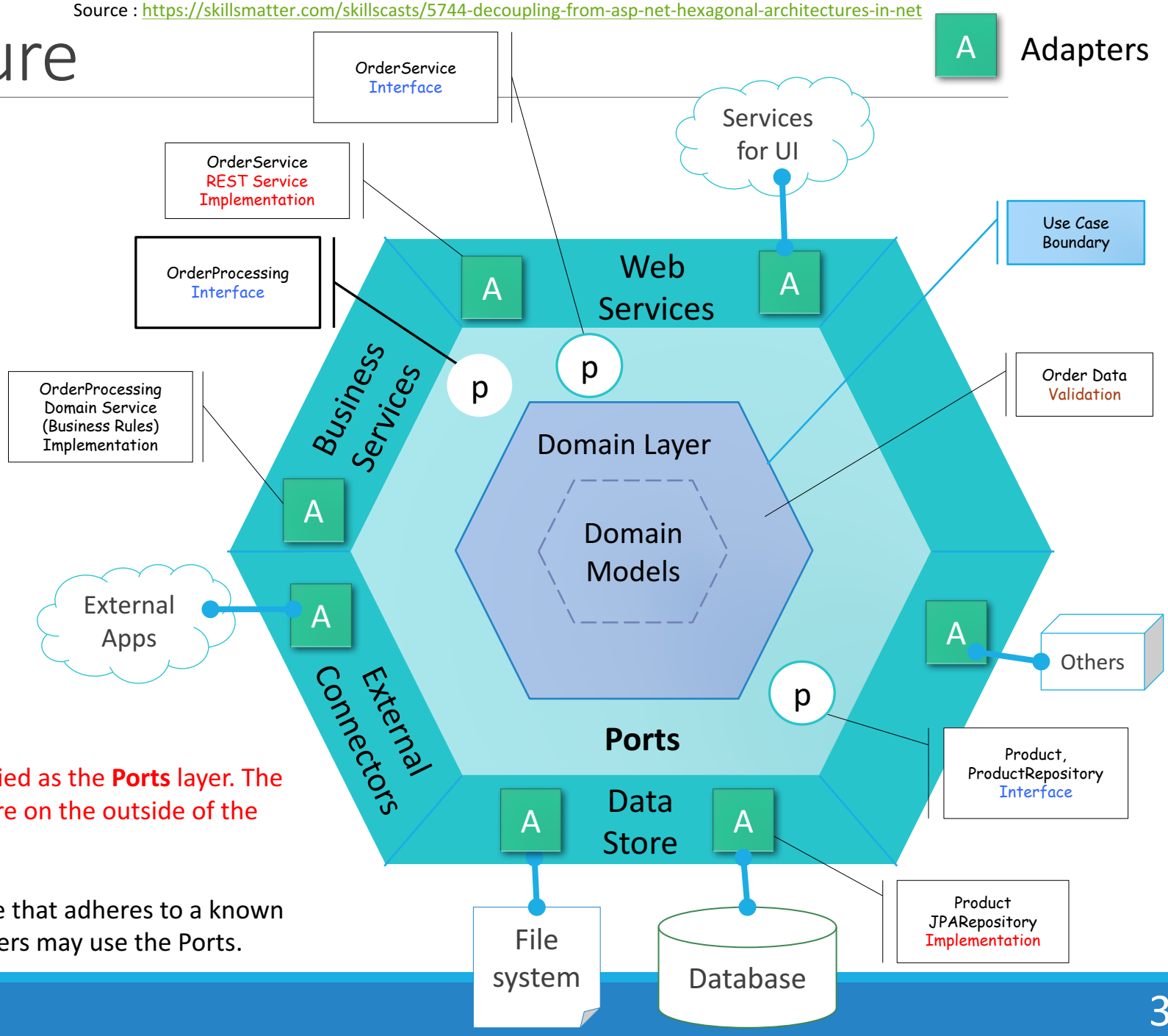
Ports and **Adapters** architecture style (Hexagonal Architecture) is a variation of layered architecture style which makes clear the separation between the **Domain Model** which contains the rules of the Application and the **adapters**, which abstract the inputs to the system and our outputs.

The advantage is the Application is decoupled from the nature of input / output device and any framework used to implement them.

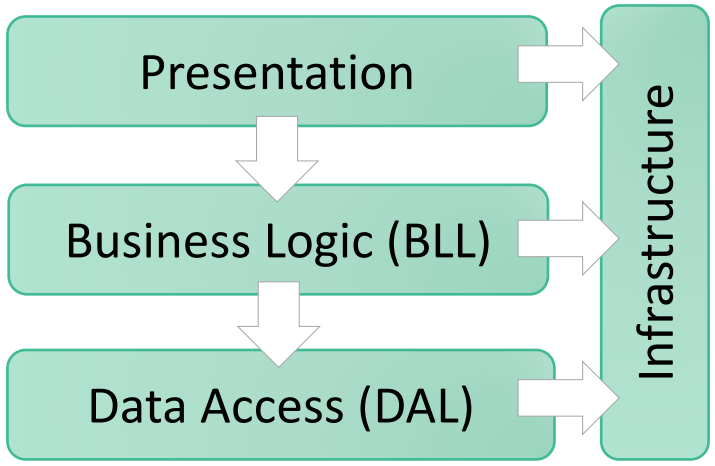
For Eg., When a client makes a REST API request the **Adapter** receives the HTTP Request and transforms that into a call into the Domain Layer and marshals the response back out to the client over HTTP. Similarly if the app needs to retrieve persisted Entity to initialize the domain it calls out to an Adapter that wraps the access to the DB.

The layer between the **Adapter** and the **Domain** is identified as the **Ports** layer. The Domain is inside the port, adapters for external entities are on the outside of the port.

The notion of a "port" invokes the OS idea that any device that adheres to a known protocol can be plugged into a port. Similarly many adapters may use the Ports.



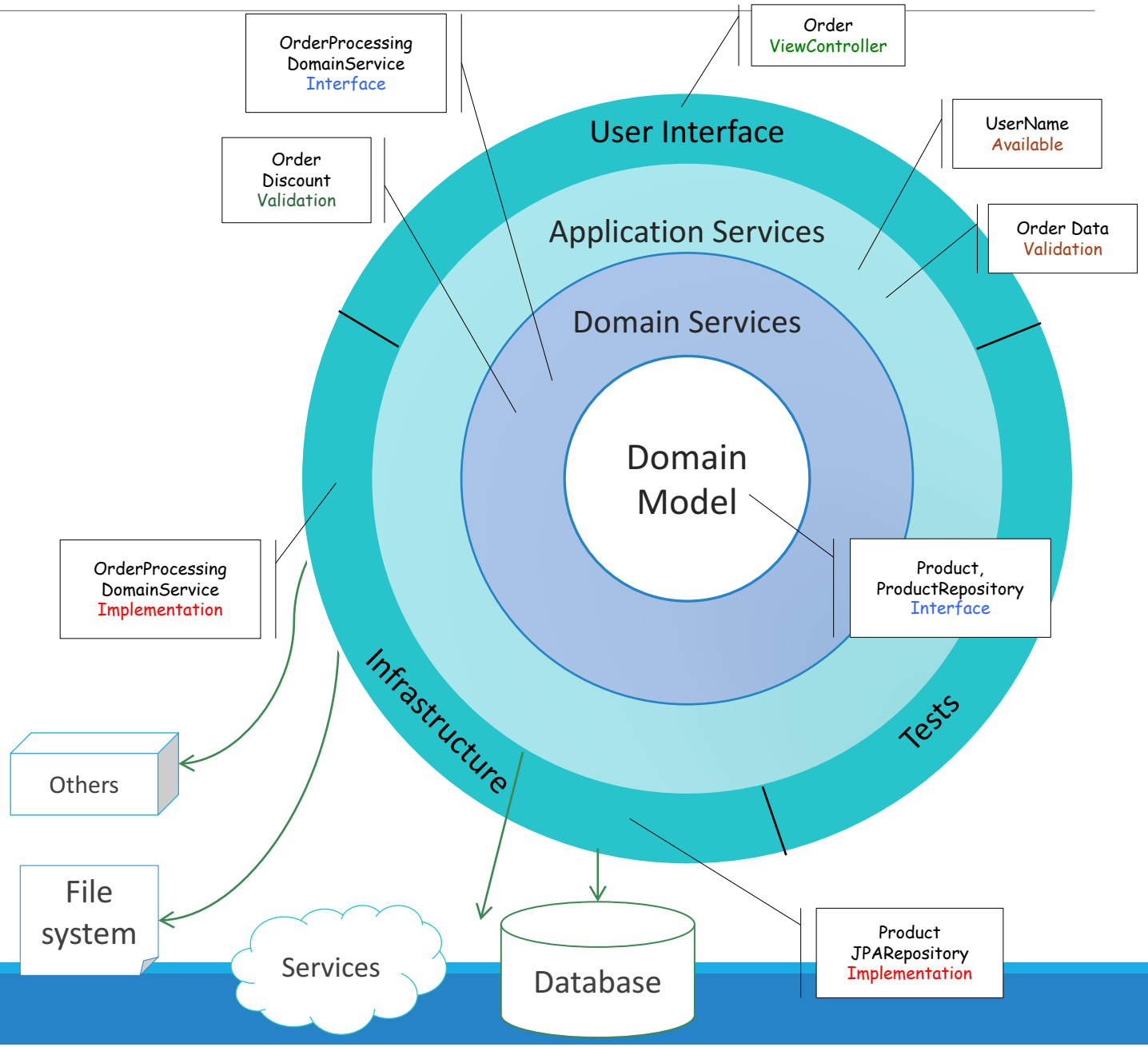
Traditional Architecture Vs. Onion Architecture + DDD



- Interface Driven
- Dependency Injection
- Auto wiring

- **Adaptable to future changes / enhancements**

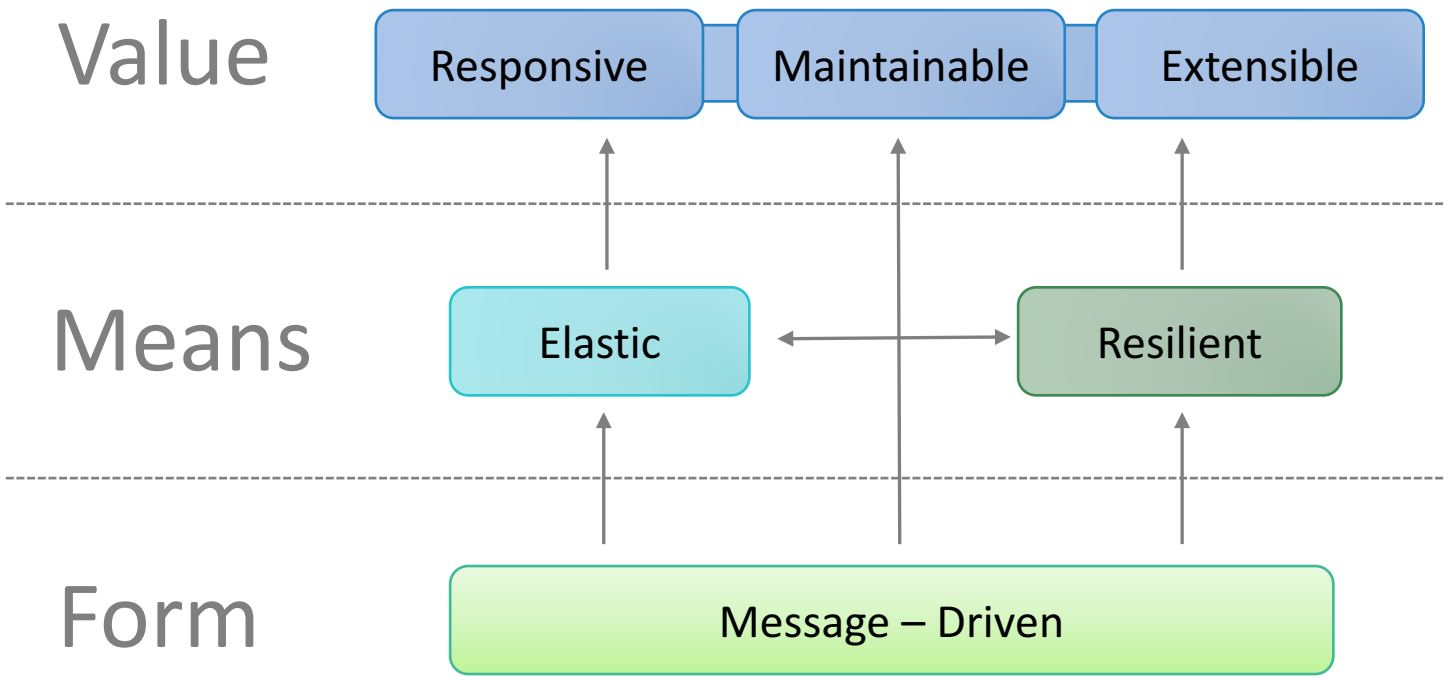
Source: Onion Architecture
By Jeffrey Palermo, Ex Microsoft



Functional Reactive Programming

Reactive (Principles)	Micro Services (Strategy)
Responsive, Resilient,	Bounded Context (DDD)
Elastic,	Event Sourcing, CQRS
Message-Driven	Eventual Consistency

Functional Reactive Programming



Principles		What it means?
Responsive	thus	React to users demand
Resilient	thus	React to errors and failures
Elastic	thus	React to load
Message-Driven	thus	React to events and messages

Reactive Extensions (Rx)

1. A ***responsive, maintainable & Extensible*** application is the goal.
2. A *responsive* application is both ***scalable (Elastic)*** and ***resilient***.
3. Responsiveness is impossible to achieve without both scalability and resilience.
4. A ***Message-Driven*** architecture is the foundation of scalable, resilient, and ultimately responsive systems.

Source: http://reactivex.io/

Functional Reactive Programming : Result = Decoupling

1. Containment of
    1. Failures
    2. Implementation Details
    3. Responsibility
2. Shared Nothing Architecture, Clear Boundaries
3. Micro Services: Single Responsibility Principle

Functional Reactive Programming : Design Patterns

Single Component Pattern

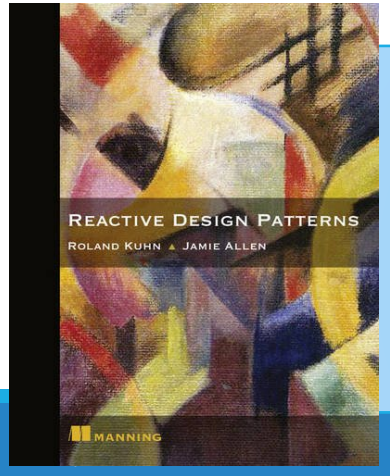
A Component shall do ONLY one thing, But do it in FULL.

Single Responsibility Principle By DeMarco : Structured Analysis & System Specification (Yourdon, New York, 1979)

Let-It-Crash Pattern

Prefer a FULL component restart to complex internal failure handling.

Candea & Fox: Crash-Only Software (USENIX HotOS IX, 2003)
Popularized by Netflix Chaos Monkey. Erlang Philosophy



Saga Pattern

Divide long-lived distributed transactions into quick local ones with compensating actions for recovery.

Pet Helland: Life Beyond Distributed Transactions CIDR 2007

ReactiveX is a combination of the best ideas from

1. the Observer pattern,
2. the Iterator pattern,
3. and functional programming

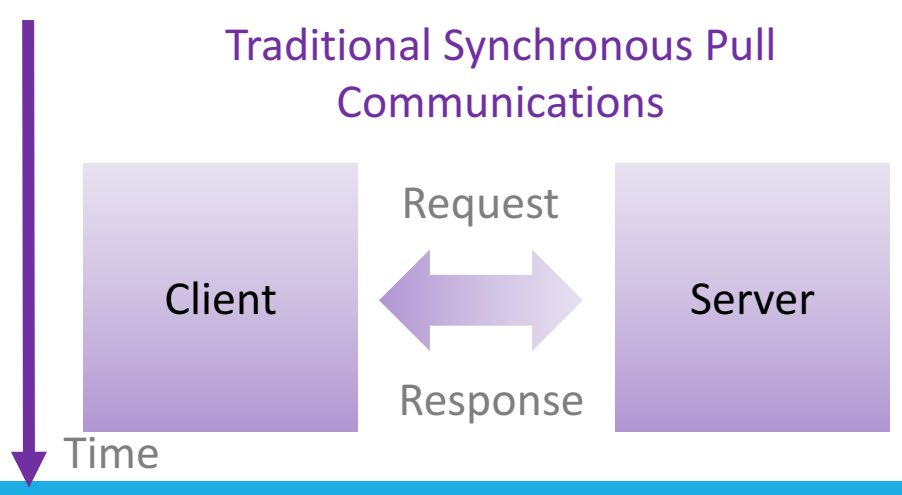
4 Building Blocks of RxJava

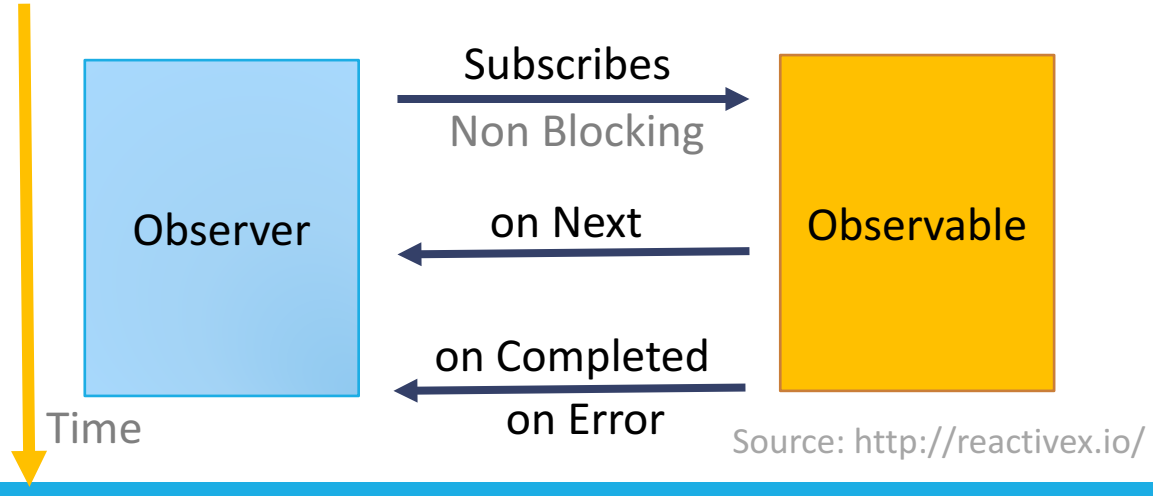
Reactive Extensions (Rx)

1 **Observable** Source of Data Stream [Sender]

2 **Observer** Listens for emitted values [Receiver]

1. The Observer subscribes (listens) to the Observable
2. Observer react to what ever item or sequence of items the Observable emits.
3. Many Observers can subscribe to the same Observable





Source: http://reactivex.io/

4 Building Blocks of Rx Java

Reactive Extensions (Rx)

3

Schedulers

Schedulers are used to manage and control concurrency.

1. observeOn: Thread Observable is executed
2. subscribeOn: Thread subscribe is executed

4

Operators

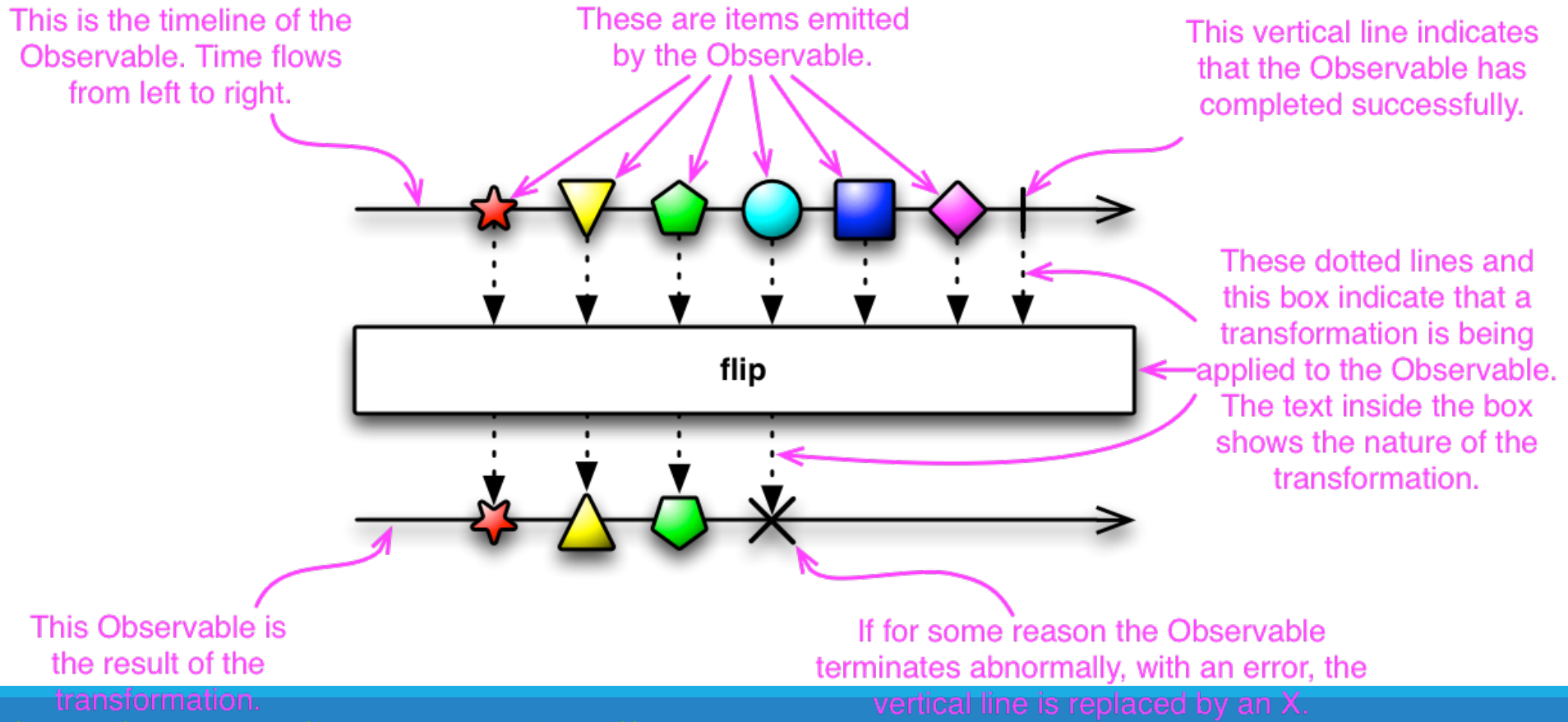
- Content Filtering
- Time Filtering
- Transformation

Operators that let you Transform, Combine, Manipulate, and work with the sequence of items emitted by Observables

Source: http://reactivex.io/

Observable Design Pattern (Marble Diagram)

- An *Observer* *subscribes* to an *Observable*.
- Then that observer reacts to whatever item or sequence of items the Observable *emits*.



Reactive Extensions (Rx)

Source: http://reactivex.io/RxJava/javadoc/index.html?rx/Observable.html | http://rxmarbles.com

Observable / Observer Design Pattern

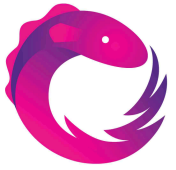
- Allows for Concurrent Operations: the **observer does not need to block** while waiting for the observable to emit values
- Observer waits to receive values when the observable is ready to emit them
- Based on push rather than pull

Source: http://reactivex.io/RxJava/javadoc/index.html?rx/Observable.html

What's missing from GOF Observer Pattern

1. The ability for the producer to signal to the consumer that there is no more data available (a for each loop on an Iterable completes and returns normally in such a case; an Observable calls its observer's onComplete method)
2. The ability for the producer to signal to the consumer that an error has occurred (an Iterable throws an exception if an error takes place during iteration; an Observable calls its observer's onError method)
3. Multiple Thread Implementations and hiding those details.
4. Dozens of Operators to handle data.

Source: http://reactivex.io/intro.html



Reactive Extensions (Rx)

Compare Iterable Vs. Observable

Observable is the asynchronous / push dual to the synchronous pull Iterable

Observables are:

- Composable: Easily chained together or combined
- Flexible: Can be used to emit:
  - A scalar value (network result)
  - Sequence (items in a list)
  - Infinite streams (weather sensor)
- Free from callback hell: Easy to transform one asynchronous stream into another

Event	Iterable (Pull)	Observable (Push)
Retrieve Data	T next()	onNext(T)
Discover Error	throws Exception	onError (Exception)
Complete	!hasNext()	onComplete()

Source: http://reactivex.io/RxJava/javadoc/index.html?rx/Observable.html

Comparison : Iterable / Streams / Observable

Java 6 – Blocking Call

```
/**
 * Iterable Serial Operations Example
 * Java 6 and 7
 *
 * @param _basket Fruit Basket Repository for Apple
 */
public void testIterable(FruitBasketRepository<Apple> _basket) {

    Iterable<Fruit> basket = _basket
                            .createAppleBasket(20)
                            .iterable();
    FruitProcessor<Apple> fp =
            new FruitProcessor<Apple>("IT");
    try {

        // Serial Operations
        for(Fruit fruit : basket) {
            fp.onNext(fruit);
        }

        fp.onComplete();

    } catch (Exception e) {
        fp.onError(e);
    }
}
```

First Class Visitor (Consumer) Serial Operations

Java 8 – Blocking Call

```
/**
 * Parallel Streams Example
 * Java 8 with Lambda Expressions
 *
 * @param _basket Fruit Basket Repository
 */
public void testParallelStream(FruitBasketRepository<Orange> _basket) {

    Collection<Fruit> basket = _basket
                                .createOrangeBasket(20)
                                .collection();
    FruitProcessor<Orange> fp =
            new FruitProcessor<Orange>("PS");

    try {

        // Parallel Operations
        basket
            .parallelStream()
            .forEach(fruit -> fp.onNext(fruit));

        fp.onComplete();
    } catch (Exception e) {
        fp.onError(e);
    }
}
```

Parallel Streams (10x Speed)
Still On Next, On Complete and On Error are Serial Operations

Rx Java - Freedom

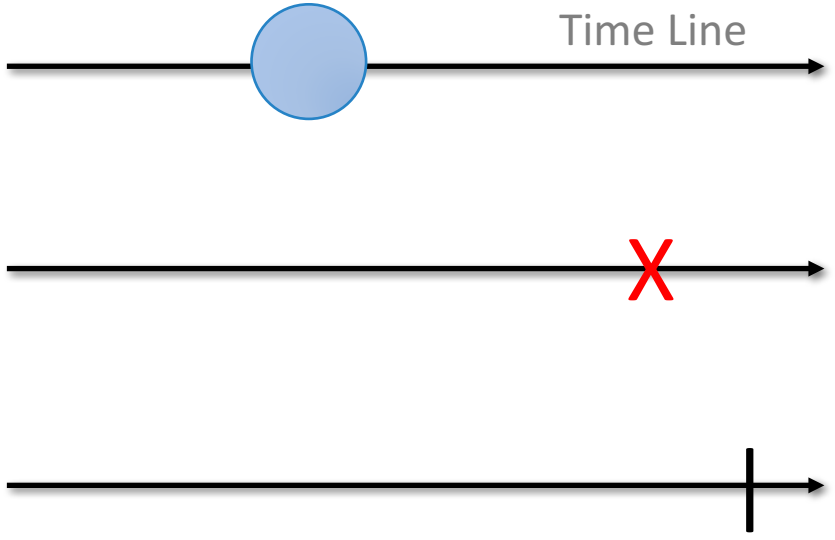
```
/**
 * Observable : Completely Asynchronous - 1
 * Functional Reactive Programming : Rx Java
 */
public void testObservable1() {

    Observable<Fruit> basket = fruitBasketObservable();
    Observer<Fruit> fp = new FruitProcessor<Apple>("Rx");

    basket
        .observeOn(Schedulers.computation())
        .subscribeOn(Schedulers.computation())
        .subscribe(
                apple -> fp.onNext(apple),
                throwable -> fp.onError(throwable),
                () -> fp.onComplete()
                );
}
```

Completely Asynchronous Operations

Observer<T> Contract

Methods:

- onNext(T)
- onError(Throwable T)
- onComplete()



onError / onComplete called exactly once

Source: http://reactivex.io/RxJava/javadoc/index.html?rx/Observable.html

Reactive Extensions (Rx)

RxJava Scheduler Details

- If you want to introduce multithreading into your cascade of Observable operators, you can do so by instructing those operators (or particular Observables) to operate on particular *Schedulers*.

- By default, an Observable and the chain of operators that you apply to it will do its work, and will notify its observers, on the same thread on which its Subscribe method is called.

- The SubscribeOn operator changes this behavior by specifying a different Scheduler on which the Observable should operate. TheObserveOn operator specifies a different Scheduler that the Observable will use to send notifications to its observers.

Source: http://reactivex.io/documentation/scheduler.html

RxJava Scheduler Threading Details

	Scheduler	Purpose
1	Schedulers.computation ( )	Meant for computational work such as event-loops and callback processing; do not use this scheduler for I/O. (useSchedulers.io( ) instead) The number of threads, by default, is equal to the number of processors
2	Schedulers.from(executor)	Uses the specified Executor as a Scheduler
3	Schedulers.immediate ( )	Schedules work to begin immediately in the current thread
4	Schedulers.io ( )	Meant for I/O-bound work such as asynchronous performance of blocking I/O, this scheduler is backed by a thread-pool that will grow as needed; for ordinary computational work, switch to Schedulers.computation( ); Schedulers.io( ) by default is a CachedThreadScheduler, which is something like a new thread scheduler with thread caching
5	Schedulers.newThread ( )	Creates a new thread for each unit of work
6	Schedulers.trampoline ( )	Queues work to begin on the current thread after any already-queued work

Source: http://reactivex.io/documentation/scheduler.html

Rx 2 Java Operator : Frequently used

	Content Filtering
1	Observable<T> filter (Predicate <? super T> predicate)
2	Observable<T> skip (int num)
3	Observable<T> take (int num)
4	Observable<T> takeLast (int num)
5	Observable<T> elementAt (int index)
6	Observable<T> distinct ()
7	Observable<T> distinctUntilChanged ()

	Time Filtering
1	Observable<T> throttleFirst (long duration, TimeUnit unit)
2	Observable<T> throttleLast (long duration, TimeUnit unit)
3	Observable<T> timeout (long duration, TimeUnit unit)

	Transformation	Use
1	Observable<T> map (Function<? super T,? extends R> mapper)	Transforms Objects
2	Observable<T> flatMap (Function<? super T, ? extends ObservableSource<? extends R>> mapper)	Transforms an Observable to another Observable
3	Observable<T> cast (Class<R> klass)	
4	Observable<GroupedObservable<K, T>> groupBy (Function<T, K> keySelector)	
5	Observable<T> merge (Iterable<? extends ObservableSource<? extends T> sources)	Merge multiple streams into a single stream

Source: http://reactivex.io/

The Problem? How Rx2 Helps!

Apple Basket

Fruit Processor

Scenario

1. A Basket Full of Apples
2. Fruit Processor capacity (limit) is 3 Apples at a time.

The Problem

1. I don't have time to wait till the Fruit Processor finishes it's job.
2. I don't mind having multiple Fruit Processors too. But that still doesn't solve the problem

What I need

1. I want to start the Process and leave.
2. Once the processing is done, then the system should inform me.
3. If something goes wrong, then the system should inform me.

Me = the calling program

Rx2Java : Examples – Handling Fruit Processor

Rx Java - Freedom

```
/**
 * Observable : Completely Asynchronous - 1
 * Functional Reactive Programming : Rx 2 Java
 */
public void testObservable1() {

    Observable<Fruit> basket = fruitBasketObservable();
    Observer<Fruit> fp = new FruitProcessor<Apple>("Rx2");

    basket
        .observeOn(Schedulers.computation())
        .subscribeOn(Schedulers.computation())
        .subscribe(
                apple -> fp.onNext(apple),
                throwable -> fp.onError(throwable),
                () -> fp.onComplete()
                );
}
```

Completely Asynchronous Operations

Entities

1. Abstract Fruit
2. Fruit (Interface)
3. Apple
4. Orange
5. Grapes
6. Mixed Fruit (Aggregate Root)

Business Layer

1. Fruit Processor (Domain Service) - Observer
2. Fruit Basket Repository
3. Fruit Basket Observable Factory

Next Section focuses more on **Operators** and how to handle business logic in Asynchronous world!

1. Rx Java Example : Observable / Observer / Scheduler Implementation
2. Rx Java Example 2 : Merge / Filters (on Weight) / Sort (on Price)

Source Code GitHub: https://github.com/meta-magic/rxjava2

Rx 2 Java : Entities & Repositories

Building Block 1

Fruit Interface

```java
public interface Fruit extends Comparable<Fruit> {

    /**
     * Returns Fruit Name
     *
     * @return String
     */
    public String name();

    /**
     * Returns the Weight
     *
     * @return int
     */
    public int weight();

    /**
     * Returns the Price
     *
     * @return int
     */
    public int getPrice();

    /**
     * Returns the Fruit Tag with Weight and Price
     *
     * @return String
     */
    public String getFruitTag();

}
```

Abstract Fruit

```java
public abstract class AbstractFruit implements Fruit {

    private int weight  = 0;
    private int price   = 0;

    /**
     * Calculates the Fruit Weight
     *
     * @param base
     * @return int
     */
    protected int calculateWeight(int base) {
        weight = (int) (Math.random() * (100 + 1));
        if(weight < base) {
            weight += base;
        }
        int price =  (int) (weight * 0.73);
        setPrice(price);
        return weight;
    }

    /**
     * Fruit Weight
     *
     * @return int
     */
    public int weight() {
        return weight;
    }

    /**
     * Set the Price
     *
     * @param _price
     */
    public void setPrice(int _price) {
        price = _price;
    }

    /**
     * Return the Price
     *
     * @return int
     */
    public int getPrice() {
        return price;
    }

    /**
     * Compares the Fruits on Prices (Ascending Order)
     */
    @Override
    public int compareTo(Fruit _fruit) {
        return Integer.compare(this.price, _fruit.getPrice());
    }
```

Fruit Repository

```java
public class FruitBasketRepository<T extends Fruit> {

    private ArrayList<Fruit> basket;

    /**
     * Creates a default basket with 20 Apples
     */
    public FruitBasketRepository() {
        basket = new ArrayList<Fruit>();
    }

    /**
     * Re Creates a New Basket of Apples
     *
     * @param int
     */
    public FruitBasketRepository<T> createAppleBasket(int limit) {
        basket.clear();
        for(int x=1; x<=limit; x++) {
            basket.add(new Apple(x));
        }
        return this;
    }

    /**
     * Creates a New Basket of Oranges
     * @param limit
     * @return
     */
    public FruitBasketRepository<T> createOrangeBasket(int limit) {
        basket.clear();
        for(int x=1; x<=limit; x++) {
            basket.add(new Orange(x));
        }
        return this;
    }

    /**
     * Creates a New Basket of Grapes
     * @param limit
     * @return
     */
    public FruitBasketRepository<T> createGrapesBasket(int limit) {
        basket.clear();
        for(int x=1; x<=limit; x++) {
            basket.add(new Grapes(x));
        }
        return this;
    }

    /**
     * Returns the Fruit Basket as a Collection
     *
     * @return Collection<Fruit>
     */
    public Collection<Fruit> collection() {
        return basket;
    }
```

Source Code GitHub: https://github.com/meta-magic/rxjava2

Rx 2 Java : Entities

Apple

```java
/**
 * Apple Entity
 *
 * @author grafkarsh
 *
 */
public class Apple extends AbstractFruit {

    private final int id;

    /**
     * Creates an Apple Object with a unique ID
     *
     * @param int
     */
    public Apple(int _id) {
        id = _id;
        calculateWeight(35);
    }

    /**
     * Fruit Name
     *
     * @return String
     */
    public String name() {
        return "Apple";
    }

    /**
     * HashCode Method. Returns ID
     *
     * @return int
     */
    public int hashCode() {
        return id;
    }

    /**
     * To String Method. Prints ID
     *
     * @return String
     */
    public String toString() {
        return "A"+id;
    }

    /**
     * Equals Method
     */
    public boolean equals(Object o) {
        try {
            Apple a = (Apple)o;
            if(id == a.id) {
                return true;
            }
        } catch (Exception e) {}
        return false;
    }

}
```

Orange

```java
/**
 * Orange Entity
 *
 * @author grafkarsh
 *
 */
public class Orange extends AbstractFruit {

    private final int id;

    /**
     * Creates an Orange Object with a unique ID
     *
     * @param int
     */
    public Orange(int _id) {
        id = _id;
        calculateWeight(25);
    }

    /**
     * Returns the Fruit Name
     *
     * @return String
     */
    public String name() {
        return "Orange";
    }

    /**
     * HashCode Method. Returns ID
     *
     * @return int
     */
    public int hashCode() {
        return id;
    }

    /**
     * To String Method. Prints ID
     *
     * @return String
     */
    public String toString() {
        return "O"+id;
    }

    /**
     * Equals Method
     */
    public boolean equals(Object o) {
        try {
            Orange a = (Orange)o;
            if(id == a.id) {
                return true;
            }
        } catch (Exception e) {}
        return false;
    }
}
```

Grapes

```java
/**
 * Grapes Entity
 *
 * @author grafkarsh
 *
 */
public class Grapes extends AbstractFruit {

    private final int id;

    /**
     * Creates an Grapes Object with a unique ID
     *
     * @param int
     */
    public Grapes(int _id) {
        id = _id;
        calculateWeight(7);
    }

    /**
     * Fruit Name
     *
     * @return String
     */
    public String name() {
        return "Grapes";
    }

    /**
     * HashCode Method. Returns ID
     *
     * @return int
     */
    public int hashCode() {
        return id;
    }

    /**
     * To String Method. Prints ID
     *
     * @return String
     */
    public String toString() {
        return "G"+id;
    }

    /**
     * Equals Method
     */
    public boolean equals(Object o) {
        try {
            Grapes a = (Grapes)o;
            if(id == a.id) {
                return true;
            }
        } catch (Exception e) {}
        return false;
    }
}
```

Source Code GitHub: https://github.com/meta-magic/rxjava2

Rx 2 Java : Apple Fruit Basket : Observable

Building Block 1

Fruit Basket Observable

```
/**
 * Creates a Fruit Basket Observable - Rx 2 Java
 *
 * @param _limit Sets the limit for the Apple Basket Observable
 * @return Observable Returns Fruit Observable
 */
public static Observable<Fruit> createAppleBasketObservable (int _limit) {

    // Apple Fruit Basket Repository call
    final FruitBasketRepository<Apple> basket = getAppleBasket(_limit);

    Observable<Fruit> obs = Observable.create(
                            new ObservableOnSubscribe<Fruit>() {

        // public void call(Subscriber<? super Fruit> observer) {
        @Override
        public void subscribe(ObservableEmitter<Fruit> observer) throws Exception {
            try {
                if (!observer.isDisposed()) {

                    basket
                        .collection()
                        // .parallelStream()
                        .forEach( fruit -> observer.onNext(fruit) );

                    observer.onComplete();
                }
            } catch (Exception e) {
                observer.onError(e);
            }
        }
    } );
    return obs;
}
```

1. This function calls the Apple Basket Repository function to load the data.
2. Observable is created using the collection from the data returned by the Repository
3. Call method checks if the Observer is NOT Disposed.
4. If Yes for Each Apple it calls the Observer to do the action = observer.onNext (fruit)
5. Once the process is completed
6. Observable will call the on Complete method in Observer.
7. If an error occurs then Observable will call the on Error method in Observer

Source Code GitHub: https://github.com/meta-magic/rxjava2

Rx 2 Java : Fruit Processor : Observer

Building Block 1

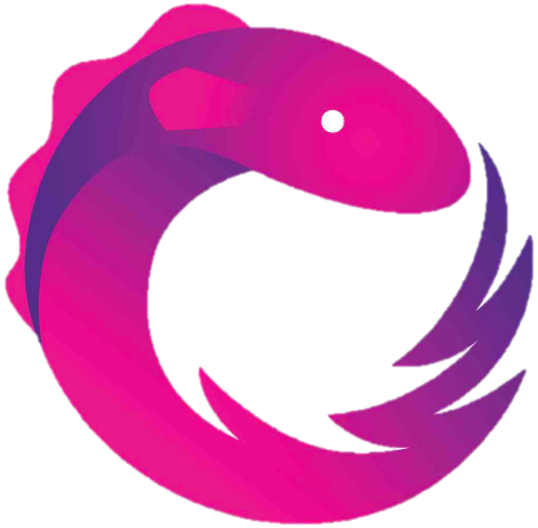
Fruit Processor Observer

```java
public class FruitProcessor<T extends Fruit>
    extends DefaultSubscriber<Fruit> implements Predicate<T>, Observer<Fruit> {

    /**
     * Rx Java 2.x Function Implementation
     * Returns TRUE if the Fruit Weight is Greater than the given Weight
     *
     * @param _fruit Sets the Fruit to check the weight
     * @return Boolean Returns TRUE if the Fruit Weight is Greater than the given Weight
     */
    @Override
    public boolean test(T _fruit) {
        return (_fruit.weight() > weight);
    }

    /**
     * Returns Weight Filter
     *
     * @return Function Returns the Filter
     */
    public Predicate<T> weightFilter() {
        return this;
    }

    /**
     * On Every Fruit Cut the Fruit into 5 pieces
     *
     * @param _fruit Handle Fruit for processing
     */
    @Override
    public void onNext(Fruit _fruit) {
        if(start) {
            startTime = System.currentTimeMillis();
            start = false;
        }
        try {
            // Time required to cut the Fruit into 5 pieces
            processFruit(_fruit);
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    /**
     * Once the Process is completed Print Stats
     */
    public void onComplete() {
        endTime = System.currentTimeMillis();
        totalTime = endTime - startTime;
        double seconds = totalTime / 1000;
        System.out.println("\nATS-"+pid+"> Fruit Process Task Completed - Total Time in Seconds = "+seconds);
        start = true;
    }

     * Throw Error if the Knife is BAD!!
    public void onError(Throwable t) {
        System.err.println("\nATS-"+pid+"> Fruit Processor : Whoops Error!! = "+t.getMessage());
```

1. Fruit Processor implements Subscriber (Observer) interface
2. On Next Method Implementation. Main Business logic is done over here. Observable will call on Next and pass the Fruit Object for processing.
3. On Complete Method. Observable will call this method once the processing is done.
4. If Any error occurred then Observable will call on Error method and pass on the Exception.
5. test() implements the Predicate for filtering. Weight is passed as a parameter in Fruit Processor Constructor.

Fruit Processor Handles the Business Logic

Source Code GitHub: https://github.com/meta-magic/rxjava2



Apple Basket

Bringing all the pieces together

Observable / Observer / Schedulers / Operators

Fruit Processor

Source Code GIT: https://meta-magic.github.io/rxjava/

Rx 2 Java Operator : Merge 3 Streams

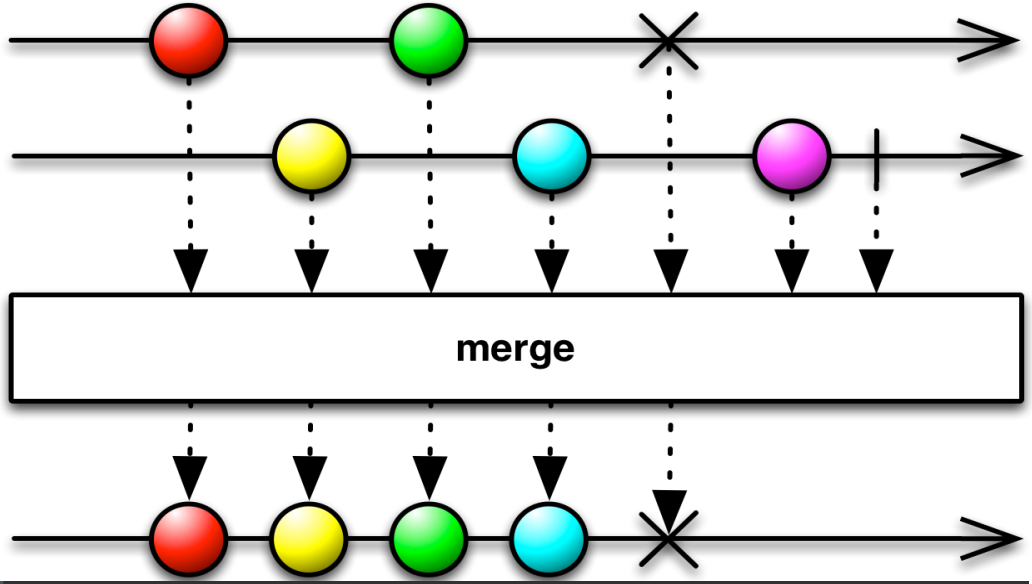
Building Block 4

Objective:

Merge Three Data Streams (Apple Fruit Basket, Orange Fruit Basket & Grapes Fruit Basket) into a Single Stream and do the Processing on that new Stream Asynchronously

Rx Example 2

```
/**
 * Merge Three Data Streams Asynchronously
 *
 * Rx 2 Java Merge Example - Asynchronous
 */
public void mergeThreeTeams() {

    Observer<Fruit> fp = fruitProcessorObserver("ME");
    Observable
        .merge( appleFruitBasketObservable(),
                orangeFruitBasketObservable(),
                grapesFruitBasketObservable()
            )
        .observeOn(Schedulers.computation())
        .subscribeOn(Schedulers.computation())
        .subscribe(
            fruit -> fp.onNext(fruit),
            throwable -> fp.onError(throwable),
            () -> fp.onComplete()
        );
}
```




```
ATS> Starting the Async Test Suite.....................

ATS> Merging Three Streams Async Test.....................
ATS> Fruit Processor (Observer) Initialized with ID = ME
ATS> Merging Three Streams Async Test Scheduled...........

ME=A1|53 ME=A2|45 ME=A3|85 ME=O1|36 ME=O2|50 ME=O3|88 ME=O4|36
ME=G1|37 ME=G2|33 ME=G3|41 ME=G4|7
ATS-ME> Fruit Process Task Completed - Total Time in Seconds = 5.0

ATS> Async Test Suite Complete ....................
```

A1 is the Apple Series, O1 is the Orange Series & G1 is the Grapes Series

Source Code GitHub: https://github.com/meta-magic/rxjava2

Rx 2 Java Operator : Merge / Filter Streams

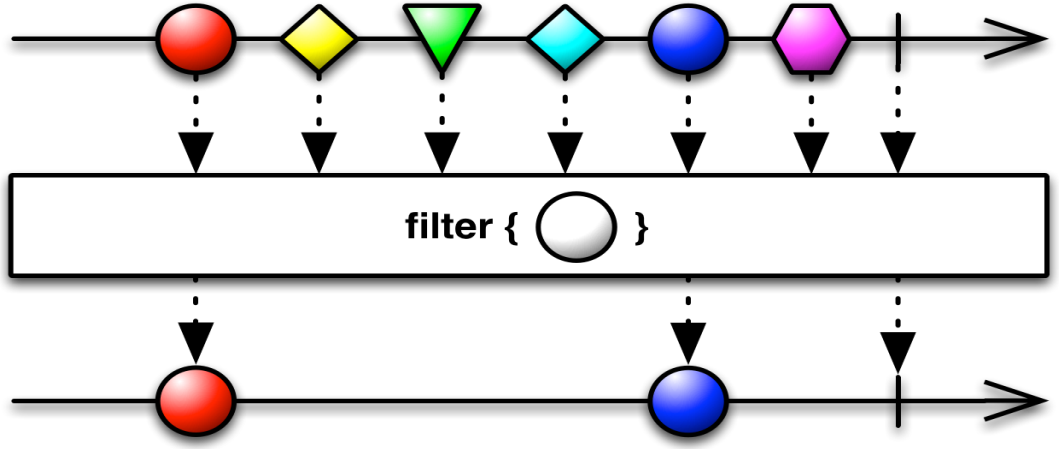
Building Block 4

Objective:

From the previous three Merged streams Filter Fruits which weighs more than 50 grams.

Rx Example 2

```
/**
 * Filter Fruits based on Weight - Rx 2 Java
 *
 * @param weight Sets the base weight for the Fruits
 */
public void filterFruitsOnWeight(final int weight) {

    Observer<Fruit> fp = fruitProcessorObserver("FI");
    Observable
        .merge( appleFruitBasketObservable(),
                orangeFruitBasketObservable(),
                grapesFruitBasketObservable()
            )
        .observeOn(Schedulers.computation())
        .subscribeOn(Schedulers.computation())
        .filter((Predicate<? super Fruit>) new Predicate<Fruit>() {
                public boolean test(Fruit _fruit) {
                    return( _fruit.weight()  > weight);
                }
        })
        .subscribe(
            fruit -> fp.onNext(fruit),
            throwable -> fp.onError(throwable),
            () -> fp.onComplete()
        );
}
```




```
ATS> Starting the Async Test Suite.....................

ATS> Merging Three Streams Async Test.....................
ATS> Fruit Processor (Observer) Initialized with ID = ME
ATS> Merging Three Streams Async Test Scheduled............

ME=A1|84 ME=A2|45 ME=A3|52 ME=01|95 ME=02|90 ME=03|92 ME=04|47
ME=G1|76 ME=G2|21 ME=G3|12 ME=G4|44
ATS-ME> Fruit Process Task Completed - Total Time in Seconds = 5.0

ATS> Merging Three Streams & Filter Async Test.....................
ATS> Filter Criteria : Weight > 60
ATS> Fruit Processor (Observer) Initialized with ID = FI
ATS> Merging Three Streams & Filter Async Test Scheduled............

FI=A1|84 FI=01|95 FI=02|90 FI=03|92 FI=G1|76
ATS-FI> Fruit Process Task Completed - Total Time in Seconds = 2.0

ATS> Async Test Suite Complete ....................
```

Source Code GitHub: https://github.com/meta-magic/rxjava2

Rx 2 Java Operator : Merge / Filter / Sort

Reactive Extensions (Rx)

Building Block 4

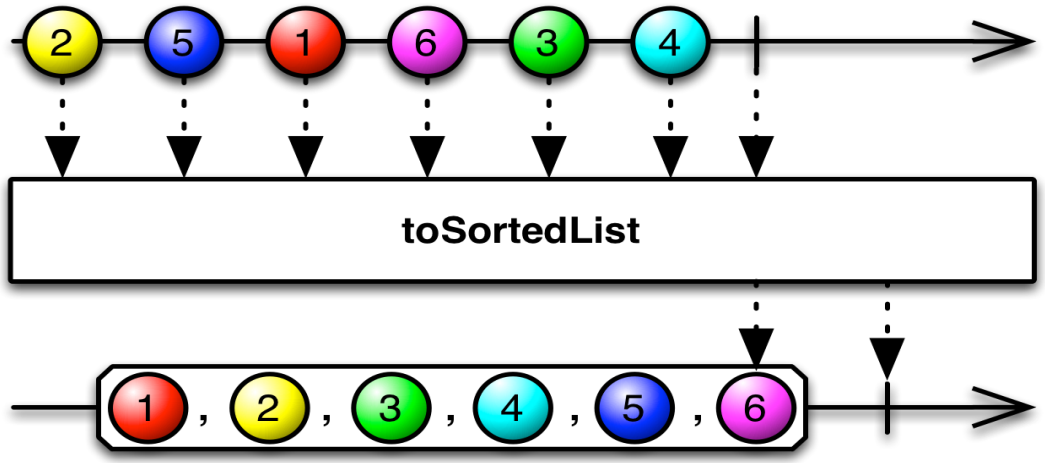
Objective:

From the previous three Merged streams Filter Fruits which weighs more than 50 grams and sort on Price (Asc).

Rx Example 2

```
/**
 * Filter, Sort on Price (Ascending) and Take (Rx 2 Java)
 *
 * @param _weight Sets the base weight for the Fruit
 * @param _take Sets How many fruits needs to be collected
 */

public void filterSortAndTake(int _weight, int _take) {
    FruitProcessor<Fruit> fp = fruitProcessor("SO", _weight);

    Observable
        .fromIterable(Observable
            .merge( appleFruitBasketObservable(),
                    orangeFruitBasketObservable(),
                    grapesFruitBasketObservable()
                  )
            .filter(fp.weightFilter())
            .toSortedList()
            .blockingGet()
        ).take(_take)
        .observeOn(Schedulers.computation())
        .subscribeOn(Schedulers.computation())
        .subscribe(
            fruit -> fp.onNext(fruit),
            throwable -> fp.onError(throwable),
            () -> fp.onComplete()
        );
}
```

Don't Use blockgingGet() in Production Code




```
ATS> Starting the Async Test Suite.....................

ATS> Merging Three Streams Async Test.....................
ATS> Fruit Processor (Observer) Initialized with ID = ME
ATS> Merging Three Streams Async Test Scheduled.....

[ME]=A1 (37gms:Rs.27) [ME]=A2 (63gms:Rs.45) [ME]=A3 (72gms:Rs.52)
[ME]=O1 (29gms:Rs.21) [ME]=O2 (84gms:Rs.61) [ME]=O3 (73gms:Rs.53)
[ME]=O4 (31gms:Rs.22) [ME]=G1 (33gms:Rs.24) [ME]=G2 (99gms:Rs.72)
[ME]=G3 (33gms:Rs.24) [ME]=G4 (53gms:Rs.38)
ATS-ME> Fruit Process Task Completed - Total Time in Seconds = 5.0

ATS> Merging Three Streams & Filter Async Test.....................
ATS> Filter Criteria : Weight > 51
ATS> Fruit Processor (Observer) Initialized with ID = FI
ATS> Merging Three Streams & Filter Async Test Scheduled.....

[FI]=A2 (63gms:Rs.45) [FI]=A3 (72gms:Rs.52) [FI]=O2 (84gms:Rs.61)
[FI]=O3 (73gms:Rs.53) [FI]=G2 (99gms:Rs.72) [FI]=G4 (53gms:Rs.38)
ATS-FI> Fruit Process Task Completed - Total Time in Seconds = 3.0

ATS> Merging Three Streams & Filter / Sort Price (Asc) Async Test...
ATS> Filter Criteria : Weight > 51 Take = 5
ATS> Fruit Processor (Observer) Initialized with ID = SO
ATS> Merging Three Streams & Filter / Sort Async Test Scheduled.....

[SO]=G4 (53gms:Rs.38) [SO]=A2 (63gms:Rs.45) [SO]=A3 (72gms:Rs.52)
[SO]=O3 (73gms:Rs.53) [SO]=O2 (84gms:Rs.61)
ATS-SO> Fruit Process Task Completed - Total Time in Seconds = 2.0

ATS> Async Test Suite Complete .....................
```

Source Code GitHub: https://github.com/meta-magic/rxjava2

Rx 2 Java Operator : Filter / Sort / FlatMap

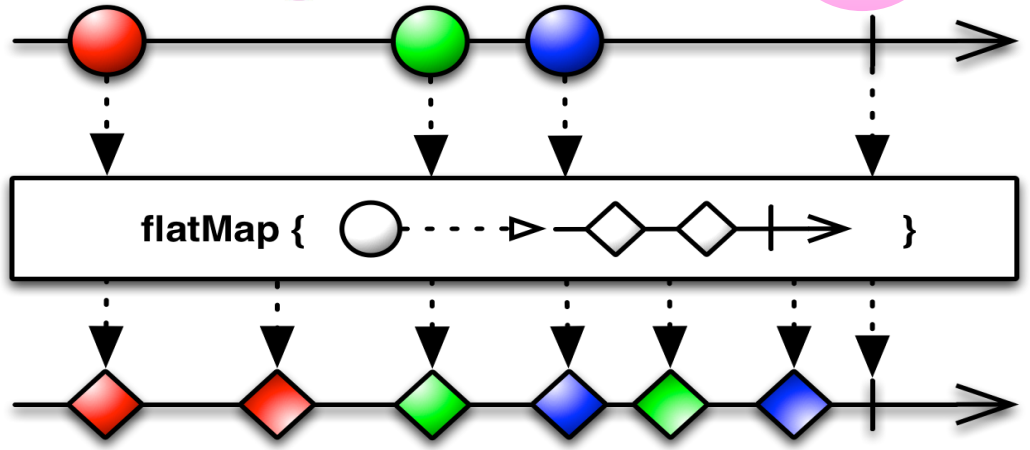
Building Block 4

Objective:

toSortedList() returns an Observable with a single List containing Fruits.
Using FlatMap to Transform Observable <List> to Observable <Fruit>

Rx Example 2

```
/**
 * Filter, Sort on Price (Ascending), Flat Map and Take
 * FlatMap is used to Transform One Observable to another
 * (Rx 2 Java)
 * @param _weight Sets the base weight for the Fruit
 * @param _take Sets How many fruits needs to be collected
 */
public void filterSortFlatMapTake(int _weight, int _take) {
    FruitProcessor<Fruit> fp = fruitProcessor("FM", _weight);

    Observable
    .merge( appleFruitBasketObservable(),
            orangeFruitBasketObservable(),
            grapesFruitBasketObservable()
        )

    .filter(fp.weightFilter())
    .toSortedList()
    .flatMapObservable(list -> Observable.fromIterable(list))
    .take(_take)

    .observeOn(Schedulers.computation())
    .subscribeOn(Schedulers.computation())
    .subscribe(
        fruit -> fp.onNext(fruit),
        throwable -> fp.onError(throwable),
        () -> fp.onComplete()
    );
}
```




```
ATS> Starting the Async Test Suite.....................

ATS> Merging Three Streams Async Test.....................
ATS> Fruit Processor (Observer) Initialized with ID = ME
ATS> Merging Three Streams Async Test Scheduled...........

[ME]=A1 [44gms:Rs.32] [ME]=A2 [69gms:Rs.50] [ME]=A3 [40gms:Rs.29]
[ME]=O1 [52gms:Rs.37] [ME]=O2 [47gms:Rs.34] [ME]=O3 [63gms:Rs.45]
[ME]=O4 [27gms:Rs.19] [ME]=G1 [60gms:Rs.43] [ME]=G2 [8gms:Rs.5]
[ME]=G3 [99gms:Rs.72] [ME]=G4 [14gms:Rs.10] [ME]=G5 [23gms:Rs.16]
ATS-ME> Fruit Process Task Completed - Total Time in Seconds = 6.0

ATS> Merging Three Streams & Filter Async Test.....................
ATS> Filter Criteria : Weight > 51
ATS> Fruit Processor (Observer) Initialized with ID = FI
ATS> Merging Three Streams & Filter Async Test Scheduled...........

[FI]=A2 [69gms:Rs.50] [FI]=O1 [52gms:Rs.37] [FI]=O3 [63gms:Rs.45]
[FI]=G1 [60gms:Rs.43] [FI]=G3 [99gms:Rs.72]
ATS-FI> Fruit Process Task Completed - Total Time in Seconds = 2.0

ATS> Merging Three Streams & Filter / Sort Price (Asc) Async Test........
ATS> Filter Criteria : Weight > 51 Take = 5
ATS> Fruit Processor (Observer) Initialized with ID = SO
ATS> Merging Three Streams & Filter / Sort Async Test Scheduled...........

[SO]=O1 [52gms:Rs.37] [SO]=G1 [60gms:Rs.43] [SO]=O3 [63gms:Rs.45]
[SO]=A2 [69gms:Rs.50] [SO]=G3 [99gms:Rs.72]
ATS-SO> Fruit Process Task Completed - Total Time in Seconds = 2.0

ATS> Merging Three Streams & Filter / Sort Price (Asc) / Flat Map Async Test.
ATS> Filter Criteria : Weight > 51 Take = 5
ATS> Fruit Processor (Observer) Initialized with ID = FM
ATS> Merging Three Streams & Filter / Sort / Flat Map Async Test Scheduled...

[FM]=O1 [52gms:Rs.37] [FM]=G1 [60gms:Rs.43] [FM]=O3 [63gms:Rs.45]
[FM]=A2 [69gms:Rs.50] [FM]=G3 [99gms:Rs.72]
ATS-FM> Fruit Process Task Completed - Total Time in Seconds = 2.0

ATS> Async Test Suite Complete .....................
```

Source Code GitHub: https://github.com/meta-magic/rxjava2

Creating Observables

Operators that originate new Observables.

- **Create** — create an Observable from scratch by calling observer methods programmatically
- **Defer** — do not create the Observable until the observer subscribes, and create a fresh Observable for each observer
- **Empty/Never/Throw** — create Observables that have very precise and limited behavior
- **From** — convert some other object or data structure into an Observable
- **Interval** — create an Observable that emits a sequence of integers spaced by a particular time interval
- **Just** — convert an object or a set of objects into an Observable that emits that or those objects
- **Range** — create an Observable that emits a range of sequential integers
- **Repeat** — create an Observable that emits a particular item or sequence of items repeatedly
- **Start** — create an Observable that emits the return value of a function
- **Timer** — create an Observable that emits a single item after a given delay

Source: http://reactivex.io/

Transforming Observables

Operators that transform items that are emitted by an Observable.

- **Buffer** — periodically gather items from an Observable into bundles and emit these bundles rather than emitting the items one at a time
- **FlatMap** — transform the items emitted by an Observable into Observables, then flatten the emissions from those into a single Observable
- **GroupBy** — divide an Observable into a set of Observables that each emit a different group of items from the original Observable, organized by key
- **Map** — transform the items emitted by an Observable by applying a function to each item
- **Scan** — apply a function to each item emitted by an Observable, sequentially, and emit each successive value
- **Window** — periodically subdivide items from an Observable into Observable windows and emit these windows rather than emitting the items one at a time

Source: http://reactivex.io/

Filtering Observables

Operators that selectively emit items from a source Observable.

- **Debounce** — only emit an item from an Observable if a particular timespan has passed without it emitting another item
- **Distinct** — suppress duplicate items emitted by an Observable
- **ElementAt** — emit only item *n* emitted by an Observable
- **Filter** — emit only those items from an Observable that pass a predicate test
- **First** — emit only the first item, or the first item that meets a condition, from an Observable
- **IgnoreElements** — do not emit any items from an Observable but mirror its termination notification
- **Last** — emit only the last item emitted by an Observable
- **Sample** — emit the most recent item emitted by an Observable within periodic time intervals
- **Skip** — suppress the first *n* items emitted by an Observable
- **SkipLast** — suppress the last *n* items emitted by an Observable
- **Take** — emit only the first *n* items emitted by an Observable
- **TakeLast** — emit only the last *n* items emitted by an Observable

Reactive Extensions (Rx)

Combining Observables

Operators that work with multiple source Observables to create a single Observable

- **And/Then/When** — combine sets of items emitted by two or more Observables by means of Pattern and Plan intermediaries
- **CombineLatest** — when an item is emitted by either of two Observables, combine the latest item emitted by each Observable via a specified function and emit items based on the results of this function
- **Join** — combine items emitted by two Observables whenever an item from one Observable is emitted during a time window defined according to an item emitted by the other Observable
- **Merge** — combine multiple Observables into one by merging their emissions
- **StartWith** — emit a specified sequence of items before beginning to emit the items from the source Observable
- **Switch** — convert an Observable that emits Observables into a single Observable that emits the items emitted by the most-recently-emitted of those Observables
- **Zip** — combine the emissions of multiple Observables together via a specified function and emit single items for each combination based on the results of this function

Source: http://reactivex.io/

Observable Utility Operators

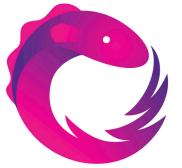
A toolbox of useful Operators for working with Observables

- **Delay** — shift the emissions from an Observable forward in time by a particular amount
- **Do** — register an action to take upon a variety of Observable lifecycle events
- **Materialize/Dematerialize** — represent both the items emitted and the notifications sent as emitted items, or reverse this process
- **ObserveOn** — specify the scheduler on which an observer will observe this Observable
- **Serialize** — force an Observable to make serialized calls and to be well-behaved
- **Subscribe** — operate upon the emissions and notifications from an Observable
- **SubscribeOn** — specify the scheduler an Observable should use when it is subscribed to
- **TimeInterval** — convert an Observable that emits items into one that emits indications of the amount of time elapsed between those emissions
- **Timeout** — mirror the source Observable, but issue an error notification if a particular period of time elapses without any emitted items
- **Timestamp** — attach a timestamp to each item emitted by an Observable
- **Using** — create a disposable resource that has the same lifespan as the Observable

Source: http://reactivex.io/

Conditional and Boolean Operators

Operators that evaluate one or more Observables or items emitted by Observables



- **All** — determine whether all items emitted by an Observable meet some criteria
- **Amb** — given two or more source Observables, emit all of the items from only the first of these Observables to emit an item
- **Contains** — determine whether an Observable emits a particular item or not
- **DefaultIfEmpty** — emit items from the source Observable, or a default item if the source Observable emits nothing
- **SequenceEqual** — determine whether two Observables emit the same sequence of items
- **SkipUntil** — discard items emitted by an Observable until a second Observable emits an item
- **SkipWhile** — discard items emitted by an Observable until a specified condition becomes false
- **TakeUntil** — discard items emitted by an Observable after a second Observable emits an item or terminates
- **TakeWhile** — discard items emitted by an Observable after a specified condition becomes false

Source: http://reactivex.io/

Mathematical and Aggregate Operators

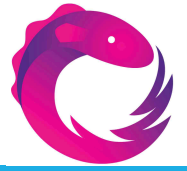
Operators that operate on the entire sequence of items emitted by an Observable

- **Average** — calculates the average of numbers emitted by an Observable and emits this average
- **Concat** — emit the emissions from two or more Observables without interleaving them
- **Count** — count the number of items emitted by the source Observable and emit only this value
- **Max** — determine, and emit, the maximum-valued item emitted by an Observable
- **Min** — determine, and emit, the minimum-valued item emitted by an Observable
- **Reduce** — apply a function to each item emitted by an Observable, sequentially, and emit the final value
- **Sum** — calculate the sum of numbers emitted by an Observable and emit this sum

Backpressure Operators

- **backpressure operators** — strategies for coping with Observables that produce items more rapidly than their observers consume them

Source: http://reactivex.io/



Rx 2 Java Operator Details

Connectable Observable Operators

Specialty Observables that have more precisely-controlled subscription dynamics

- **Connect** — instruct a connectable Observable to begin emitting items to its subscribers
- **Publish** — convert an ordinary Observable into a connectable Observable
- **RefCount** — make a Connectable Observable behave like an ordinary Observable
- **Replay** — ensure that all observers see the same sequence of emitted items, even if they subscribe after the Observable has begun emitting items

Operators to Convert Observables

- **To** — convert an Observable into another object or data structure

Error Handling Operators

Operators that help to recover from error notifications from an Observable

- **Catch** — recover from an onError notification by continuing the sequence without error
- **Retry** — if a source Observable sends an onError notification, re-subscribe to it in the hopes that it will complete without error

Source: http://reactivex.io/

Rx Java 1 to Rx Java 2 Migration

- *onCompleted* became *onComplete* — without the trailing d
- *Func1* became *Function*
- *Func2* became *BiFunction*
- *CompositeSubscription* became *CompositeDisposable*
- *limit* operator has been removed. Use *take* in RxJava 2

Rx Java 1 to Rx Java 2 Migration

This is not a comprehensive list, but a most used list.

Rx Java 1.0 Package = rx.*	
Object	**Method Call**
import rx.Observable; Observable.operator	Observer call(Observer)
import rx.*; Observer	**void onCompleted**
Subscriber implements Observer	
Import rx.Observable; Observerable.onSubscribe http://reactivex.io/RxJava/1.x/javadoc/	void **call(Observer)** Observer > Subscriber onNext() **onCompleted()** onError() - **isUnSubscribed()**

Rx Java 2.0 Package = io.reactivex.*	
Object	**Method Call**
import io.reactivex.*; ObservableOperator	Subscriber apply(Subscriber)
import io.reactivex.*; Observer	**void onComplete**
No Subscriber Class in v2.0	
Import io.reactivex.* ObservableOnSubscriber http://reactivex.io/RxJava/2.x/javadoc/	Void **subscribe(ObservableEmitter)** throws Exception Emitter > ObservableEmitter onNext() **onComplete()** onError() - **IsDisposed()**

Rx Java 1 to Rx Java 2 Migration

This is not a comprehensive list, but a most used list.

Rx Java 1.0 Package = rx.*	
Object	**Method Call**
import rx.functions.*; Func1	T call(T)
import rx.Observable;	Observable filter(Func1)
Observable	Observable first()
	Observable flatMap(Func1)
	void forEach(Action1)
	Observable from(Future) Observable from(Iterable) Observable from(Array)
	Observable(GroupedObservable) groubBy(Func1)
	Observable groupJoin(Observable)

Rx Java 2.0 Package = io.reactivex.*	
Object	**Method Call**
io.reactivex.functions.*; Function1	T apply(T)
import io.reactivex.*;	Observable filter(Predicate)
Observable	Single first()
	Observable flatMap(Function)
	Disposable forEach(Consumer)
	Observable fromFuture(Future) Observable fromIterable(Iterable) Observable fromArray(Array)
	Observable(GroupedObservable) groubBy(Function)
	Observable groupJoin(ObservableSource)

http://reactivex.io/RxJava/1.x/javadoc/

http://reactivex.io/RxJava/2.x/javadoc/

Rx Java 1 to Rx Java 2 Migration

This is not a comprehensive list, but a most used list.

Rx Java 1.0 Package = rx.*	
Object	**Method Call**
import rx.Observable; **Observable**	Observable Join(Observable, Func1)
	Observable<Boolean> all(Func1)
	Observable ~~amb(Observable….)~~
	~~asObservable()~~
	Observable collect(Func0, Action2)
	Observable contact(Observable)
	Observable concatMap(Func1)
	Observable contains()

Rx Java 2.0 Package = io.reactivex.*	
Object	**Method Call**
import io.reactivex.*; **Observable**	Observable Join(ObservableSource, Function)
	Single<Boolean> all(Predicate)
	Observable amb(Iterable<ObservableSource>)
	Single collect(java.util.concurrent.Callable)
	Observable concat(ObservableSource)
	Observable concatMap(Function)
	Single contains()

http://reactivex.io/RxJava/1.x/javadoc/

http://reactivex.io/RxJava/2.x/javadoc/

Rx Java 1 to Rx Java 2 Migration

This is not a comprehensive list, but a most used list.

Rx Java 1.0 Package = rx.*	
Object	**Method Call**
import rx.Observable;	Observable ~~countLong()~~
Observable	static Observable create(Action1<Emitter>, Emitter.backpressure)
	~~static Observable create(-----)~~
	Observable debounce(Func1)
	Observable distinct(Func1)
	~~Observable exists(Func1)~~
	Observable first()

Rx Java 2.0 Package = io.reactivex.*	
Object	**Method Call**
import io.reactivex.*;	
Observable	static Observable create(ObservableOnSubscriber)
	Observable debounce(Function)
	Observable distinct(Function)
	Single first(T)

http://reactivex.io/RxJava/1.x/javadoc/

http://reactivex.io/RxJava/2.x/javadoc/

Thank You

Araf Karsh Hamid
email : araf.karsh@metamagic.in
Social : arafkarsh | Skype, Facebook, LinkedIn, Twitter, G+
Phone : +91.999.545.8627 | http://www.metamagicglobal.com